NumPy

November 30, 2024

1 Learn NumPy

NumPy is the fundamental package for scientific computing in Python. It is a Python library that provides a multidimensional array object, various derived objects (such as masked arrays and matrices), and an assortment of routines for fast operations on arrays, including mathematical, logical, shape manipulation, sorting, selecting, I/O, discrete Fourier transforms, basic linear algebra, basic statistical operations, random simulation and much more.

NumPy arrays have a fixed size at creation, unlike Python lists (which can grow dynamically). Changing the size of an ndarray will create a new array and delete the original.

The elements in a NumPy array are all required to be of the same data type, and thus will be the same size in memory. The exception: one can have arrays of (Python, including NumPy) objects, thereby allowing for arrays of different sized elements.

NumPy arrays facilitate advanced mathematical and other types of operations on large numbers of data. Typically, such operations are executed more efficiently and with less code than is possible using Python's built-in sequences.

A growing plethora of scientific and mathematical Python-based packages are using NumPy arrays; though these typically support Python-sequence input, they convert such input to NumPy arrays prior to processing, and they often output NumPy arrays. In other words, in order to efficiently use much (perhaps even most) of today's scientific/mathematical Python-based software, just knowing how to use Python's built-in sequence types is insufficient - one also needs to know how to use NumPy arrays.

sequence size and speed are particularly important in scientific computing

NumPy's main object is the homogeneous multidimensional array. It is a table of elements (usually numbers), all of the same type, indexed by a tuple of non-negative integers. In NumPy dimensions are called axes. For example, the array for the coordinates of a point in 3D space, [1, 2, 1], has one axis. That axis has 3 elements in it, so we say it has a length of 3. In the example pictured below, the array has 2 axes. The first axis has a length of 2, the second axis has a length of 3.

1.0.1 Difference between Numpy ad standard Python

NumPy arrays have a fixed size at creation, unlike Python lists (which can grow dynamically). Changing the size of an ndarray will create a new array and delete the original

The elements in a NumPy array are all required to be of the same data type, and thus will be the same size in memory. The exception: one can have arrays of (Python, including NumPy) objects, thereby allowing for arrays of different sized elements.

NumPy arrays facilitate advanced mathematical and other types of operations on large numbers of data. Typically, such operations are executed more efficiently and with less code than is possible using Python's built-in sequences.

A growing plethora of scientific and mathematical Python-based packages are using NumPy arrays; though these typically support Python-sequence input, they convert such input to NumPy arrays prior to processing, and they often output NumPy arrays.

[328]:
```
[[1., 0., 0.],
 [0., 1., 2.]]
```

[328]:
```
[[1.0, 0.0, 0.0], [0.0, 1.0, 2.0]]
```

NumPy's array class is called ndarray. It is also known by the alias array. Note that numpy.array is not the same as the Standard Python Library class array.array, which only handles one-dimensional arrays and offers les functionality. The more important attributes of an ndarray object are 1. ndarray.n:im the number of axes (dimensions) of the a

2. ndarray.shape: the dimensions of the array. This is a tuple of integers indicating the size of the array in each dimension. For matrix with n rows and mcolumns, shapewillbe(n,m). Theleng th oft hesha petup le ist herefo ret henumbr of axes, nd

3. . ndarray.:ize the total number of elements of the array. This is equal to the product of the elements of s

4. pe. ndarray:dtype an object describing the type of the elements in the array. One can create or specify dtype's using standard ython types. Additionally NumPy provides types of its own. numpy.int32, numpy.int16, and numpy.float64 re some

5. amples. ndarra:.itemsize the size in bytes of each element of the array. For example, an array of elements of type float64asitemsize 8 (=64/8), while one of type complex32 has itemsize 4 (=32/8). It is equivalent to ndray.dtype

6. itemsize.: ndarray.data the buffer containing the actual elements of the array. Normally, we won't need to use this attriute because we will access the elements in an array using indexing facilities.rray

[330]:
```
import numpy as np
np.__version__
```

[330]:
```
'1.26.4'
```

[331]:
```
list1 = [1,2,3,4,5]
list1
```

[331]:
```
[1, 2, 3, 4, 5]
```

1.0.2 Basic Attributes of the ndarray Class

[333]:
```
list2 = ['mukesh', 28, 10000000, [1,2,3,4,5,6], True]
list2
```

[333]:
```
['mukesh', 28, 10000000, [1, 2, 3, 4, 5, 6], True]
```

This doesnot work with massive amount of data as it will become very slow. so NumPy was introduces wich is every fast but allows to manipulate similar kind of data.

[335]:
```
import numpy as np
```

[1]:
```
#help(np.ndarray)
```

Attribute	Description
Shape	A tuple that contains the number of elements (i.e., the length) for each dimension (axis) of the array.
Size	The total number elements in the array.
Ndim	Number of dimensions (axes).
nbytes	Number of bytes used to store the data.
dtype	The data type of the elements in the array.

[338]:
```
np1 = np.array([0,1,2,3,4,5,6,7,8,9]) #creatin NumPy array
np1  #display the values of NumPy array
```

[338]:
```
array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
```

[339]:
```
type(np1)
```

[339]:
```
numpy.ndarray
```

[340]:
```
np1.shape
```

[340]:
```
(10,)
```

[341]:
```
np1.size
```

[341]:
```
10
```

[342]:
```
np1.ndim
```

[342]:
```
1
```

[343]:
```
np1.nbytes
```

[343]: 40

[344]:
```
np1.dtype
```

[344]: dtype('int32')

1.0.3 Data Types

dtype	Variants	Description
int	int8, int16, int32, int64	Integers
uint	uint8, uint16, uint32, uint64	Unsigned (nonnegative) integers
bool	Bool	Boolean (True or False)
float	float16, float32, float64, float128	Floating-point numbers
complex	complex64, complex128, complex256	Complex-valued floating-point numbers

1.0.4 3 Ways to Creating NumPy Arrays

1st method: via python list

[349]:
```
mylist = [1,2,3,4,5]
mylist_from_array = np.array(mylist)
mylist_from_array
```

[349]: array([1, 2, 3, 4, 5])

[350]:
```
mylist_from_array * 10
```

[350]: array([10, 20, 30, 40, 50])

2nd Method: via Python Tuples

[352]:
```
# creation via Python Tuple
mytuple = (1,2,3,4,5,6,7,8,9,10)
np.array(mytuple)
```

[352]: array([1, 2, 3, 4, 5, 6, 7, 8, 9, 10])

3rd method: via NumPy arane method

[354]:
```
np.arange(10)
```

[354]: array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])

[355]:
```
np.arange(10, 23)
```

[355]:
```
array([10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22])
```

[356]:
```
np.arange(10,23,5)
```

[356]:
```
array([10, 15, 20])
```

The following example demonstrates how to use the dtype attribute to generae arrays of intger-, foat-, and complex-valued elements:

[358]:
```
npp = np.array([1,2,3,4], dtype=np.int16)
print(npp)
```

```
[1 2 3 4]
```

[359]:
```
npp1 = np.array([1,2,3,4,5,], dtype=np.float32)
print(npp1)
```

```
[1. 2. 3. 4. 5.]
```

[360]:
```
npp2 = np.array([1,2,3,4,5,6], dtype=np.complex64)
npp2
```

[360]:
```
array([1.+0.j, 2.+0.j, 3.+0.j, 4.+0.j, 5.+0.j, 6.+0.j], dtype=complex64)
```

Once a NumPy array is created, its dtype cannot be changed

[362]:
```
data = np.array([1, 2, 3], dtype=np.float32)
data
```

[362]:
```
array([1., 2., 3.], dtype=float32)
```

[363]:
```
data.dtype
```

[363]:
```
dtype('float32')
```

[364]:
```
data = np.array(data, dtype=np.int32)
data.dtype
```

[364]:
```
dtype('int32')
```

[365]:
```
data
```

[365]:
```
array([1, 2, 3])
```

It ca also be changed using the astype method of the ndarray class

[367]:
```
data = np.array([1, 2, 3], dtype=np.float32)
data
```

[367]:
```
array([1., 2., 3.], dtype=float32)
```

[368]:
```
data.astype(np.int32)
```

[368]:
```
array([1, 2, 3])
```

[369]:
```
 d1 = np.array([1, 2, 3], dtype=float)
 d2 = np.array([1, 2, 3], dtype=complex)
d1 + d2
```

[369]:
```
array([2.+0.j, 4.+0.j, 6.+0.j])
```

[370]:
```
 (d1 + d2).dtype
```

[370]:
```
dtype('complex128')
```

Real and Imaginary Parts

[372]:
```
data = np.array([1, 2, 3], dtype=complex)
data
```

[372]:
```
array([1.+0.j, 2.+0.j, 3.+0.j])
```

[373]:
```
data.real
```

[373]:
```
array([1., 2., 3.])
```

[374]:
```
data.imag
```

[374]:
```
array([0., 0., 0.])
```

length of the array

[376]:
```
len(np.arange(10,23))
```

[376]:
```
13
```

[377]:
```
np.arange(10,23).size
```

[377]:
```
13
```

1.0.5 Difference between Python ad NumPy Data Structure

[379]:
```
mytuple * 3
```

[379]:
```
(1,
 2,
 3,
 4,
 5,
 6,
 7,
 8,
 9,
 10,
 1,
 2,
 3,
 4,
 5,
 6,
 7,
 8,
 9,
 10,
 1,
 2,
 3,
 4,
 5,
 6,
 7,
 8,
 9,
 10)
```

[380]:
```
np.array(mytuple) * 3
```

[380]:
```
array([ 3,  6,  9, 12, 15, 18, 21, 24, 27, 30])
```

1.0.6 NumPy shape

[382]:
```
np1.shape #showing the shape of array, dimentions and/or numer of values in␣
↪array(in the current case as it is just one dimentaional array)
```

[382]:
```
(10,)
```

[383]:
```
np2 = np.array([[1,2,3,4,5],[6,7,8,9,10]])
np2.shape
```

[383]:
```
(2, 5)
```

[384]:
```
np2
```

[384]:
```
array([[ 1,  2,  3,  4,  5],
       [ 6,  7,  8,  9, 10]])
```

[385]:
```
a.ravel() #returnsthearray,flattened
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
Cell In[385], line 1
----> 1 a.ravel()

NameError: name 'a' is not defined
```

[ ]:
```
np2 = np.arange(10)  #creating NumPy array with arange() with 10 values
np2
```

[ ]:
```
np3 = np.arange(0,10,2) #usin arange steps option with value 2
np3
```

[ ]:
```
np5 = np.zeros((2,10)) # creatin NumPy array with 2 dimentional with 10 items␣
 ↪in each dimention
np5
```

[ ]:
```
np6 = np.full((10), 6) #NumPy full(0 function to create NumPy single␣
 ↪dimentional array with 10 values and all values are 6
np6
```

[ ]:
```
np7 = np.full((2, 10), 6) #NumPy full(0 function to create NumPy two␣
 ↪dimentional array with 10 values and all values are 6
np7
```

[ ]:
```
a = np.empty([2, 3, 2])
a
```

[ ]:
```
#converting list to NumPy array
mylist = [1,2,3,4,5]
np8 = np.array(mylist)
np8
```

[ ]:
```
#accessing ay specific item of NumPy array
np8[0]
```

[ ]:
```
np8[2]
```

[ ]:
```
#checkin the datatype of NumPy array
np8.dtype
```

1.0.7 Two dimentional arrays

[ ]:
```
my_array = np.arange(35)
my_array.shape = (7,5)
my_array
```

[ ]:
```
my_array[2]
```

[ ]:
```
my_array[-2]
```

[ ]:
```
my_array[6,2]
```

[ ]:
```
my_array[6][2]
```

1.0.8 Three Dimentional Array

[ ]:
```
my3darray = np.arange(70)
my3darray.shape=(2,7,5)
my3darray
```

1.0.9 Basic Indexing and Slicing

[ ]:
```
np1 = np.array([1,2,3,4,5,6,7,8,9])
```

[ ]:
```
np1[1:5]
```

[ ]:
```
np1[1:]
```

[ ]:
```
np1[-3:-1]
```

[ ]:
```
np1[1:5:2] #with stepping 2
```

[ ]:
```
np1[::2] #prints start to end with stepping 2
```

[ ]:
```
np1[::3] #prints start to end with stepping 3
```

[ ]:
```
arr = np.arange(10)
arr
```

[ ]:
```
arr[5]
```

[ ]:
```
arr[5:8]
```

[ ]:
```
arr[5:8] = 12
```

[ ]:
```
arr
```

An important first distinction from Python's built-in lists is that array slices are views on the original array. This means that the data is not copied, and any modifications to the view will be reflected in the source array.

[ ]:
```
arr_slice = arr[5:8]
arr_slice
```

[ ]:
```
arr_slice[1] = 12345
```

[ ]:
```
arr
```

The "bare" slice [:] will assign to all values in an array:

[ ]:
```
arr_slice[:] = 64
```

[ ]:
```
arr
```

[ ]:
```
#slicing multi dimentional array
np2 = np.array([[1,2,3,4,5], [6,7,8,9,10]])
np2[0:1,1:3]
```

[ ]:
```
np2[0:2,1:3]
```

[ ]:
```
 arr3d = np.array([[[1, 2, 3], [4, 5, 6]], [[7, 8, 9], [10, 11, 12]]])
```

[ ]:
```
arr3d
```

[ ]:
```
 arr3d[0]
```

1.1 Copy Vs View for NumPy

[ ]:
```
import numpy as np
np1 = np.array([0,1,2,3,4,5])
```

[ ]:
```
#creating view
np2= np1.view()
np1 #original NP1
```

[ ]:
```
np2 #Original NP2
```

[ ]:
```
np1[0] = 41
```

[ ]:
```
np1 #original NP1
```

[ ]:
```
np2 #Original NP2
```

notice that the original and view both changed. It means whe we changing the original, the view also changes

Aother example

[ ]:
```
 rng = np.random.default_rng(seed=1701)  # seed for reproducibility
x1 = rng.integers(10, size=6)  # one-dimensional array
x2 = rng.integers(10, size=(3, 4))  # two-dimensional array
x3 = rng.integers(10, size=(3, 4, 5))  # three-dimensional array
x2
```

[ ]:
```
 x2_sub = x2[:2, :2]
x2_sub
```

[ ]:
```
 #copy function
np1 = np.array([0,1,2,3,4,5])
```

Now if we modify this subarray, we'll see that the original array is changed!

[ ]:
```
x2_sub[0, 0] = 99
x2_sub
```

[ ]:
```
x2
```

1.1.1 Creating a Copy

[ ]:
```
np1 = np.array([0,1,2,3,4,5])
np2 = np1.copy()
```

[ ]:
```
np1 #original NP1
```

[ ]:
```
np2 #original NP1
```

[ ]:
```
np1[0] = 41
```

[ ]:
```
np1 #original NP1
```

[ ]:
```
np2 #original NP1
```

notice that when we change the origional but the copy has not changed. It simply idicates the copy is a seperate copuy fothe origial and has no links witht eh origional

[ ]:
```
# now try to change the copy version to check if the original changes
```

[ ]:
```
np1 = np.array([0,1,2,3,4,5])
```

[ ]:
```
np2 = np1.copy()
```

[ ]:
```
np2
```

[ ]:
```
np1 #original NP1
```

[ ]:
```
np2[0] = 41
```

[ ]:
```
np1 #original NP1
```

[ ]:
```
np2 #original NP1
```

since copy is completely diferent array it only chcnges and it doesnot reflect the original array

1.2 Another example

[ ]:
```
x2 = rng.integers(10, size=(3, 4))  # two-dimensional array
x2_sub_copy = x2[:2, :2].copy()
x2_sub_copy
```

If we now modify this subarray, the original array is not touched

[ ]:
```
x2_sub_copy[0, 0] = 42
```

[ ]:
```
x2_sub_copy
```

[ ]:
```
x2
```

1.2.1 Indexing with Slices

[ ]:
```
arr
```

[ ]:
```
arr[1:6]
```

[ ]:
```
arr2d = np.array([[1, 2, 3], [4, 5, 6], [7, 8, 9]])
```

[ ]:
```
arr2d
```

[ ]:
```
arr2d[:2]
```

[ ]:
```
arr2d[:2, 1:]
```

[ ]:
```
lower_dim_slice = arr2d[1, :2]
```

[ ]:
```
lower_dim_slice
```

[ ]:
```
lower_dim_slice.shape
```

[ ]:
```
 arr2d[:2, 2]
```

[ ]:
```
arr2d[:, :1]
```

[ ]:
```
arr2d[:2, 1:] = 0
```

[ ]:
```
 arr2d
```

[ ]:
```
arr2d[1, :2]
```

[ ]:
```
my_vector = np.array([-15,-4,0,3,21,37,105])
my_vector
```

[ ]:
```
my_vector[0]
```

[ ]:
```
my_vector[1]
```

[ ]:
```
my_vector[305]
```

[ ]:
```
305 % 7
```

[ ]:
```
my_vector[305 % 7]
```

check how it gives the 4th element of the my_vector array above

[ ]:
```
my_vector.size
```

Boolean Indexing

[ ]:
```
 names = np.array(["Bob", "Joe", "Will", "Bob", "Will", "Joe", "Joe"])
```

[ ]:
```
names
```

[ ]:
```
data = np.array([[4, 7], [0, 2], [-5, 6], [0, 0], [1, 2], [-12, -4], [3, 4]])
```

[ ]:
```
data
```

[ ]:
```
names == "Bob"
```

[ ]:
```
 data[names == "Bob"]
```

[ ]:
```
 data[names == "Bob", 1:]
```

```
[ ]: data[names == "Bob", 1]
```

```
[ ]: names != "Bob"
```

```
[ ]: ~(names == "Bob")
```

```
[ ]: data[~(names == "Bob")]
```

```
[ ]: cond = names == "Bob"
```

```
[ ]: data[~cond]
```

```
[ ]: mask = (names == "Bob") | (names == "Will")
```

```
[ ]: mask
```

```
[ ]: data[mask]
```

```
[ ]: data[data < 0] = 0
```

```
[ ]: data
```

```
[ ]: data[names != "Joe"] = 7
```

```
[ ]: data
```

1.2.2 Another example

```
[ ]: my_vector = np.array([-17,-3,0,4,7,19,109])
```

```
[ ]: my_vector
```

```
[ ]: zero_mod_7_mask = 0 == (my_vector % 7)
     zero_mod_7_mask
```

```
[ ]: sub_array =  my_vector[zero_mod_7_mask]
     sub_array
```

```
[ ]: sub_array[sub_array > 0]
```

1.2.3 NumPy logical Operator

Fancy Indexing

```
[ ]: arr = np.zeros((8, 4))
```

```
[ ]: for i in range(8):
         arr[i] = i
```

```
[ ]: arr
```

```
[ ]:  arr[[4, 3, 0, 6]]
```

```
[ ]:  arr[[-3, -5, -7]]
```

```
[ ]:  arr = np.arange(32).reshape((8, 4))
```

```
[ ]: arr
```

```
[ ]:  arr[[1, 5, 7, 2], [0, 3, 1, 2]]
```

```
[ ]:  arr[[1, 5, 7, 2]][:, [0, 3, 1, 2]]
```

```
[ ]: arr.shape
```

1.2.4 NumPy universal Functions

```
[ ]: np1 = np.array([0,1,2,3,4,5,6,7,8,9])
     np1
```

```
[ ]: #sqrt function
     np.sqrt((np1))
```

```
[ ]: #abslute value function
     np1= np.array([-3,-2-1,0,1,2,3,4,5,6,7,8,9])
     np.absolute((np1))
```

```
[ ]: #exponent fujnction
     np.exp((np1))
```

```
[ ]: #min and max function
     np.max((np1))
```

```
[ ]: np.min((np1))
```

```
[ ]: #dot product calculation
     l1 = [1,2,3]
     l2 = [4,5,6]
     a1 = np.array((l1))
     a2 = np.array((l2))
     print("dot product via list")
     dot =0
```

```
for i in range(len(l1)):
    dot += l1[i] * l2[i]
print(dot)

print("dot poroduct via NumPy array")
dot  = np.dot(a1,a2)
print(dot)
```

[ ]:
```
#linspae function
np.linspace(5,15,9)
```

[ ]:
```
mylinspace = np.linspace(5,15,9, retstep=True)
mylinspace
```

[ ]:
```
mylinspace[1]
```

[ ]:
```
#zeros() function
np.zeros(6)
```

[ ]:
```
#one function
np.ones(6)
```

[ ]:
```
np.zeros((5,5))
```

[ ]:
```
np.zeros((5,4,3))
```

1.2.5 NumPy DataTypes

[ ]:
```
np.zeros(11, dtype='int64')
```

[ ]:
```
np.zeros(11)
```

[ ]:
```

```

Transposing Arrays and Swapping Axes

[ ]:
```
arr = np.arange(15).reshape((3, 5))
```

[ ]:
```
arr
```

[ ]:
```
 arr.T #returnsthe array,transposed
```

[ ]:
```
 arr = np.array([[0, 1, 0], [1, 2, -2], [6, 3, 2], [-1, 0, -1], [1, 0, 1]])
```

[ ]:
```
arr
```

[ ]:
```
np.dot(arr.T, arr)
```

[ ]:
```
arr.T @ arr
```

[ ]:
```
arr
```

[ ]:
```
arr.swapaxes(0, 1)
```

1.2.6 Pseudorandom Number Generation

[ ]:
```
samples = np.random.standard_normal(size=(4, 4))
```

[ ]:
```
samples
```

[ ]:
```
from random import normalvariate
```

[ ]:
```
N = 1_000_000
```

1.2.7 NumPy Reshaping

[ ]:
```
np1 = np.array([1,2,3,4,5,6,7,8,9,10,11,12])
np1
```

[ ]:
```
np1.shape
```

[ ]:
```
np2 = np1.reshape(3,4)
np2
```

[ ]:
```
np2.shape
```

[ ]:
```
np3 = np1.reshape(2,3,2)
np3
```

[ ]:
```
np3.shape
```

[ ]:
```
np4 = np3.reshape(-1)
np4
```

[ ]:
```
np4.shape
```

1.2.8 Another example

[ ]:
```
grid = np.arange(1, 10).reshape(3, 3)
grid
```

Note that for this to work, the size of the initial array must match the size of the reshaped array, and in most cases the reshape method will return a no-copy view of the initial array.

[ ]:
```
x = np.array([1,2,3])
x.reshape((1,3))
```

[ ]:
```
x.reshape((3,1))
```

[ ]:
```
x[np.newaxis, :]
```

[ ]:
```
 x[:, np.newaxis]
```

1.2.9 Stacking together different arrays

[ ]:
```

```

1.2.10 Iterate through NumPy array

[ ]:
```
np1 = np.array([1,2,3,4,5,6,7,8,91,10,11,12])
```

[ ]:
```
np1
```

np2 = np.array([[1,2,3,4,5],[6,7,8,9,10]]) np2

[ ]:
```
for x in np.nditer(np2):
    print(x)
```

1.2.11 Concatenation of Arrays

[ ]:
```
x = np.array([1,2,3])
y = np.array([3,2,1])
np.concatenate([x,y])
```

[ ]:
```
z = np.array([3,5,6,8,67])
np.concatenate([x,y,z])
```

[ ]:
```
grid = np.array([[1, 2, 3],[4, 5, 6]])
np.concatenate([grid,grid])
```

[ ]:
```
 # concatenate along the second axis (zero-indexed
 np.concatenate([grid, grid], axis=1)
```

For working with arrays of mixed dimensions, it can be clearer to use the np.vstack (vertical stack) and np.hstack (horizontal stack) functions

[ ]:
```
 # vertically stack the arrays
 np.vstack([x, grid])
```

```
[ ]: # horizontally stack the arrays
     y = np.array([[99],[99]])
     np.hstack([grid, y])
```

1.2.12 Splitting of Arrays

The opposite of concatenation is splitting, which is implemented by the functions np.split, np.hsplit, and np.vsplit. For each of these, we can pass a list of indices giving the split points

```
[ ]: x = [1, 2, 3, 99, 99, 3, 2, 1]
     x1, x2, x3 =  np.split(x, [3,5])
```

```
[ ]: x1
```

```
[ ]: x2
```

```
[ ]: x3
```

1.3 Sorting NumPy array

```
[ ]: np1 = np.array([5,1,78,9,1,5786,2,4,7,0])
     np.sort(np1)
```

```
[ ]: np2 = np.array(['Zakir','John', 'Tina', 'Aaron'])
     np.sort(np2)
```

```
[ ]: np3 = np.array([True, False, True, False, True, True, False])
     np.sort(np3)
```

```
[ ]: np4 = np.array([[3,7,1,5,89],[3,8,90,2,90]])
     np.sort(np4)
```

```
[ ]: grid = np.arange(16).reshape((4, 4))
     grid
```

```
[ ]: upper, lower = np.vsplit(grid, [2])
     print(upper)
     print(lower)
```

```
[ ]: left, right = np.hsplit(grid, [2])
     print(left)
     print(right)
```

1.3.1 NumPy Searching

```
[ ]: np1 = np.array([1,2,3,4,5,3,6,68678,6,78,87,78])
     print(np1)
     print(np.where(np1==3))
```

```
[ ]: a = np.where(np1==3)[0]
     print(a)
```

```
[ ]: print(np1[a[0]])
```

```
[ ]: ## prints even numbers
     y = np.where(np1 % 2 == 0)
     print(np1)
     print(y)
     print(y[0])
```

```
[ ]: ## prints odd numbers
     y = np.where(np1 % 2 == 1)
     print(np1)
     print(y)
     print(y[0])
```

Filtering NumPy Arrays with Boolean Index lists

```
[ ]: np1 = np.array([1,2,3,4,5,6,7,8,9,10])
     x = [True, True, False, False, True, False, False, False, False, False]
```

```
[ ]: print(np1)
     print(np1[x])
```

```
[ ]: filtered = []
     for thing in np1:
         if thing % 2 == 0:
             filtered.append(True)
         else:
             filtered.append(False)
     print(np1)
     print(filtered)
     new = np1[filtered]
     print(new)
```

```
[ ]: filtered = []
     for thing in np1:
         if thing > 5:
             filtered.append(True)
         else:
```

```
        filtered.append(False)
print(np1)
print(filtered)
new = np1[filtered]
print(new)
```

[ ]:
```
filtered = np1 % 2 ==0
print(np1)
print(filtered)
new = np1[filtered]
print(new)
```

[ ]:
```
filtered = np1 > 5
print(np1)
print(filtered)
new = np1[filtered]
print(new)
```

[ ]:
```

```